Trabajo Fin de Grado

Validación y refinamiento de un modelo estadístico para la clasificación y humanización de anticuerpos

Autor

David Luna Cerralbo

Directores

Pierpaolo Bruscolini

Sergio Pérez Gaviro

Universidad de Zaragoza
2020

Índice

1. Introducción.

En los últimos años, hemos asistido a un aumento de interés en el campo de experimentación con anticuerpos terapéuticos. El proceso de humanización de anticuerpos es fundamental para desarrollar estos últimos. Este proceso, sin embargo, es todavía muy lento y laborioso. En este trabajo, pondremos a prueba un modelo estadístico para generar secuencias de anticuerpos humanos, que ha sido desarrollado previamente en [1] y [2].

Los anticuerpos son unas proteínas que forman parte del sistema inmune. Su función es neutralizar virus o bacterias que han sido previamente reconocidas como extrañas. Si observamos su estructura (Figura 1) veremos que consiste en cuatro cadenas polipeptídicas, dos cadenas pesadas y dos ligeras, idénticas entre sí, unidas por puentes disulfuro. Podemos distinguir 2 regiones en cada una de estas 4 cadenas: la región constante y la región variable; esta última cumplirá la función de reconocimiento antigénico, y será la que tenga interés para nosotros.

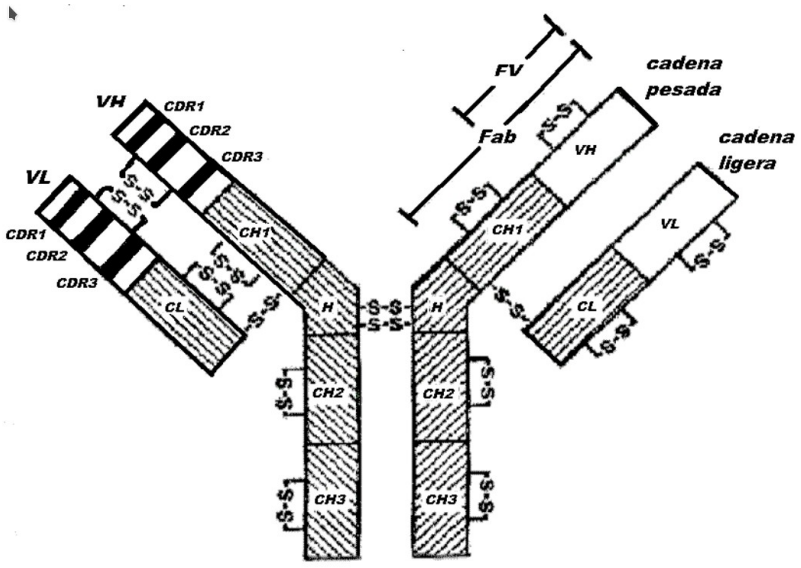


Figura 1: Estructura de un anticuerpo.[4]

Si profundizamos más en la estructura de la región variable notamos la existencia de segmentos con una elevada variabilidad. Estos segmentos, de los cuales tendremos tres en cada cadena, se denominan regiones hipervariables o 'determinantes de la complementariedad' (Complementarity Determining Regions, CDR) con el antígeno, son responsables, mayoritariamente, de las interacciones antígeno-anticuerpo. El resto de la región variable se denomina 'región marco' (Framework Region, FR) y tiene, fundamentalmente, un papel estructural. Para que la región variable realice su función de reconocimiento antigénico de manera correcta ha de acoplarse al agente extraño. Por ello, debe existir una complementariedad física y química entre la parte variable del anticuerpo y las proteínas que exhiba en su superficie el agente extraño.

Para obtener anticuerpos contra virus, bacterias, etc, utilizables en humanos, sin infectar de forma directa a miembros de nuestra especie, se infecta a un animal, comúnmente un ratón, y luego se extraen los anticuerpos que haya generado para protegerse. Sin embargo, no se pueden inyectar esos anticuerpos directamente en humanos, pues se producirían fuertes respuestas de rechazo por ser de origen extraño. Hay que realizar un proceso previo de humanización de esos anticuerpos. Este proceso consiste en transferir las CDR de los anticuerpos de animal a las regiones variables de los anticuerpos humanos, a la vez que se cambian también un cierto número de aminoácidos en las FR humanas, para hacerlas compatibles con las CDR de origen animal. De esta forma, se evita que nuestro organismo los reconozca como extraños pero se mantiene intacta la especificidad contra el virus o bacteria. Lamentablemente, no existe un criterio unívoco para saber las mutaciones exactas en las 'regiones marcos' FR, y el proceso estandar ('CDR grafting') procede en esta tarea por ensayo y error.

En [1] se propuso un protocolo de humanización basado en un modelo estadístico de la distribución de los aminoácidos en las secuencias humanas, que tiene en cuenta las correlaciones entre aminoácidos en distintas posiciones, asumiendo una distribución gaussiana. El protocolo da buenos resultados, sin embargo, se desconoce cuáles son los aspectos claves que determinan su fiabilidad, ni cómo modificarlo para mejorar los resultados. El objetivo de este trabajo será comprender cómo de fiable es la modelización gaussiana de la región variable de un anticuerpo. Para ello, generaremos nuevas bases de datos de forma artificial y controlada, aunque comenzaremos previamente validando los protocolos que permiten elegir λ y Ω del modelo. Explicaremos en la sección 2.1 qué representan y cómo se hallan, en la formulación del modelo, estos dos parámetros. A continuación, discutiremos cómo generar una distribución gaussiana adaptada a las necesidades de representar secuencias reales. Explicaremos qué criterio usamos para asociar un vector en $\mathbb{R}^{(20L)}$ a una secuencia de L aminoácidos y, finalmente, qué pruebas realizamos para evaluar la fiabilidad del criterio utilizado en [1] para fijar los parámetros. En la sección 3 se mostrarán los resultados preliminares, y en la sección 4 se resumirán las conclusiones alcanzadas, discutiendo los desarrollos futuros. Exceptuando la sección 2.1, todos los contenidos mencionados representan contribuciones originales. Los apéndices completan los métodos, y recogen los códigos desarrollados autónomamente para esta investigación.

Para realizar todo el trabajo se necesitará un lenguaje de programación que sea eficiente y adecuado. Se escoge el lenguaje 'Julia'. Se trata de un lenguaje de programación desarrollado precisamente para computación genética y científica. Tiene también un desempeño muy bueno, acercándose al de lenguajes estáticamente compilados como C pero que también nos permite llamar funciones de Python. Además, destacamos que partiremos de programas tomados de [1], que están programados usando este lenguaje.

2. Métodos.

2.1. Modelo Gaussiano Multivariante.

En esta sección vamos a ver el Modelo gaussiano multivariante publicado en [1] y en el cual nos apoyaremos. Según este modelo, cada una de las cadenas de la base de datos están distribuidas de la forma:

$$f(x; \mu, \Sigma) = N(\mu, \Sigma) \tag{1}$$

donde x es un vector real de dimensión 20L, relacionado con una secuencia de longitud L según se explica en la sección 2.1.1.
Los parámetros μ y Σ se estiman a partir de un archivo de secuencias que nos servirá como base de datos 'de aprendizaje', a través de un enfoque de inferencia bayesiana que precisa otros dos parámetros, Ω y λ, que se discutirán en las secciones 2.1.2 y 2.1.3. Ω se encargará de corregir la posible falta de independencia de las secuencias, y λ que pesará la contribución de la distribución de probabilidad previa frente a la contribución fenomenológica obtenida de la base de datos.

2.1.1. Codificación de las secuencias.

Hemos mencionado en la sección anterior que para estimar Σ y μ necesitamos una base de datos 'de aprendizaje'. Las secuencias de la base de datos experimental, formada por exactamente 1309 cadenas y tomada de [3], habían sido alineadas siguiendo el sistema AHo de alineamiento, lo que implica una longitud de 298 aminoácidos para cada cadena, 149 para VH y 149 para VL.
El objetivo de usar este sistema de alineamiento es que todas las cadenas tengan la misma longitud, facilitando el trabajar con ellas y alineado los aminoácidos estructuralmente y funcionalmente análogos. Para igualarlas todas en cuanto a tamaño, tendremos que incorporar los llamados 'gaps'. Los 'gaps' no son aminoácidos reales, los representaremos con '-', y la posición en la que aparecerán en nuestras cadenas depende del sistema de alineamiento usado. En nuestro caso aparecerán agrupados y en zonas concretas de las cadenas.
En la Figura 2 podemos ver un ejemplo del uso de los gaps en el alineamiento, así como un ejemplo de secuencias VH y VL alineadas y unidas con el protocolo AHo:

(a) Ejemplo de alineamiento y uso de gaps en el caso de nucleótidos.[6]

```
   EVQLLEW-GAGLLKPSETLSLTCAVYG-GSFSG-----YYWSWIRQPPGKGLEWIGEINH----SGSTNYNPS
LKSRVTISVDTSKNQFSLKLSSVTAADTAVYYCAVYYYDSSGY--------------QTGDAFDIWGQGTMVTVSS
--ELTQP-PSVSVSPGQTASITCSGDK-LGDK-------YAYWYQQQSGQSPVLVIYH--------DAKRPSGIPE
RFSGSNSG--NTATLTISGTQAMDEADYYCQAWDN----------------------NNVVFGGGTKLTVL
```

(b) Ejemplo de secuencia real.

Figura 2: Ejemplos aclaratorios.

Nos apoyaremos en el modelo Gaussiano Multivariante tomado de [1] y de [2] para reproducir los resultados obtenidos con la base de datos experimental mencionada, y para estudiar las cadenas generadas de manera artificial según explicado en las secciones 2.2 y 2.4.

No podemos trabajar con datos de tipo *Char* (Letras), por lo que seguiremos el modelo ya usado en [2]. Por cada letra de nuestra cadena, tendremos un bloque de 20 *Floats* (Flotantes) con un valor de 0 o 1, ya que nuestro diccionario tendrá una longitud de 20 aminoácidos y el gap. Cada componente de este bloque de longitud 20 representará un aminoácido. De esta forma, tendremos un vector de longitud veinte enteramente de 0's salvo un 1, colocado en la misma posición que ocupa el aminoácido que representa en la lista de aminoácidos. Si tenemos un gap, nos encontraremos un vector de 20 ceros. Es decir, si queremos representar la letra 'A', tendremos un vector de longitud 20, con 19 ceros y un 1 en la primera posición. Podemos verlo de forma esquemática en, Figura (3):

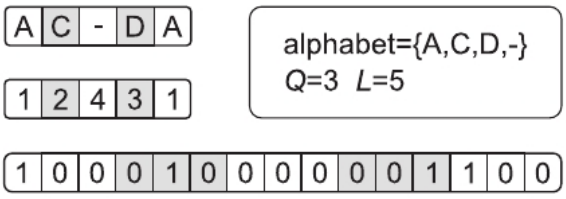


Figura 3: Ejemplo de codificación de una cadena, [2]. Para que sea más fácil la visualización del proceso, hemos reducido nuestro diccionario a 3 aminoácidos más el gap, por lo que nuestros bloques serán de longitud 3 y no 20. También hemos reducido la longitud de nuestra cadena de 298 a 5.

El código para realizar este proceso se encuentra en la función *asciitobinary*(), que hemos tomado de [1].

2.1.2. Covariancia y Media fenomenológica.

Hemos comentado anteriormente, al definir Σ y μ, que necesitaríamos 2 parámetros adicionales y que uno de ellos sería Ω. Este es un parámetro que va a intervenir a la hora de calcular el peso de cada cadena, w_m, de la siguiente forma:

$$w_m = \frac{1}{\sum_{l=1}^{M} \theta(\alpha_{lm} - L\Omega)} \tag{2}$$

donde θ representa la función escalón y siendo α_{lm} la similitud entre 2 cadenas cuando las comparamos aminoácido a aminoácido, exceptuando los gaps, y que expresamos como:

$$\alpha_{lm} = \sum_{i=1}^{L} \left(\delta_{lm} + (1 - \delta_{lm})(1 - \delta_{A_i^l,'-'})\delta_{A_i^l A_i^m} \right) \tag{3}$$

donde A_i^m representa el aminoácido i de la cadena m.
Definir Ω y los pesos de cada secuencia de esta manera nos permite evitar sesgos relacionados con el muestreo de las secuencias; para inferir el modelo gaussiano, las secuencias deberían ser estadísticamente independientes, cosa que los experimentos no nos proporcionan. De esta manera, pesaremos poco las secuencias que sean muy similares entre sí.
Con los pesos introducidos en la Ecuación 2, definimos la media y la covarianza empíricas, cuya expresión será:

$$\bar{x}_i = \frac{1}{M_e} \sum_{m=1}^{M} x_i^m w_m \quad (i = 1, ..., L) \tag{4}$$

$$C_{ij} = \frac{1}{M_e} \sum_{m=1}^{M} (x_i^m - \bar{x}_i)(x_j^m - \bar{x}_j) w_m \tag{5}$$

Definimos también M_e, que representará el número efectivo de cadenas de nuestra base, como:

$$M_e = \sum_{m=1}^{M} w_m \tag{6}$$

siendo M el número de cadenas total que tenemos en nuestra base de datos.

2.1.3. El modelo.

Una vez definidos los pesos de cada cadena, siguiendo los pasos de [1] y [2], supondremos que los parámetros Σ y μ siguen una distribución $Wishart\ Normal\ Inversa$, tal que:

$$p^{pr}(\mu, \Sigma) = N\left(\mu|\eta, \frac{\Sigma}{k}\right) IW(\Sigma|\Lambda, \nu) \tag{7}$$

Esta será nuestra distribución de probabilidad previa para Σ y μ, en la cuál apoyaremos parte de nuestro modelo.

Sabiendo esto, y usando el Teorema de Bayes, podemos calcular la distribución a posteriori de los parámetros Σ y μ dada una base de datos X. Hallamos:

$$p^{post}(\mu, \Sigma|X) \propto p(X|\mu, \Sigma)p^{pr}(\mu, \Sigma) = N\left(\mu|\eta', \frac{\Sigma}{k'}\right) IW(\Sigma|\Lambda', \nu') \tag{8}$$

de donde deducimos:

$$< \mu >_{post} = \lambda\eta + (1-\lambda)\bar{x} \tag{9}$$

$$< \Sigma >_{post} = \lambda U + (1-\lambda)\bar{C} + \lambda(1-\lambda)(\bar{x}-\eta)(\bar{x}-\eta)^T \tag{10}$$

Podemos ver el desarrollo completo en [1].
Los algoritmos para calcular tanto la probabilidad previa como la probabilidad posterior de Σ y μ los tomaremos de [1].
Únicamente queda definir λ. Este parámetro no sólo nos permite regularizar Σ, si no que también nos indica qué porcentaje de probabilidad previa queremos introducir a la hora de construir nuestro modelo estadístico. Es decir, λ nos permite regular qué peso le damos a las variables fenomenológicas y cuánto a la probabilidad previa. En concreto, en [1] se eligieron los valores λ=0.1, Ω=0.489 como la mejor opción, con la base de datos 'de aprendizaje' disponible.

2.2. Generación de secuencias a partir de un modelo gaussiano dado.

Nuestra estrategia para valorar la validez de los criterios utilizados en [1] se basa en generar bases de datos artificiales de diferente naturaleza, empezando por el caso más sencillo, de generación a partir de una probabilidad gaussiana, para ver si el método Modelo Gaussiano Multivariante permite reconstruir la gaussiana de partida.

2.2.1. Generación de secuencias con una distribución gaussiana dada.

Antes de presentar los métodos distintos con los que vamos a generar las secuencias, destacar que cualquier elección de μ y Σ sería posible, siempre que tuviesen anticorrelaciones tales que los bloques de 20 x 20 en Σ o de longitud 20 en μ tengan una sola componente considerable como 1 y el resto 0. Ante este escenario, tomamos Σ y μ del modelo desarrollado en [1] obtenido a a partir de la base de datos 'de aprendizaje'.

El primer método consiste en hacer uso de las librerías prediseñadas de Julia para este tipo de usos. En concreto, vamos a usar la librería *Distribuciones* de Julia. Esta librería nos permite generar un objeto *distribucion* que, al llamarlo, nos devolverá un vector de la longitud que nosotros queramos y distribución preestablecida. Para más información sobre el funcionamiento de esta librería de Julia y para ver el código ir al Anexo 5.1: *Distribuciones.jl*.
En nuestro caso, queremos que nos devuelva un vector x de longitud 5960 (la longitud en Char de la cadena es de 298, al pasar a nuestro sistema de numeración numérico multiplicamos por la longitud de nuestro diccionario de aminoácidos, en este caso 20) cuyas

componentes estén distribuidas de forma normal, tal que:

$$f(x|\mu,\Sigma) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} exp(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)) \tag{11}$$

Una vez hayamos obtenido un vector de longitud 5960, aplicaremos el método explicado en la sección 2.1.1 para obtener un vector de longitud 298 y tipo Char, que será el formato final en el que almacenaremos nuestra cadena.

En el segundo paso vamos a hacer uso del algoritmo de Metrópolis[1], del cual se puede consultar su desarrollo en el Anexo 5.2: *Algoritmo de Metrópolis: Funcionamiento e Implementación.*
Necesitamos este segundo método debido a que el procedimiento anterior genera vectores independientes de acuerdo a la distribución escogida; sin embargo, las secuencias de las bases de datos no son estadísticamente independientes, pero sí lo son las que generamos con las librerías de Julia. Al aplicar Monte Carlo[2] sobre las distintas semillas que generemos con las librerías de Julia, romperemos esa independencia estadística que existe entre las semillas del primer método, acercándonos a la dependencia estadística real que existe entre las cadenas del archivo 'de aprendizaje'.
Una vez hayamos generado N semillas usando las librerías de Julia el procedimiento consistirá en realizar barridos de Monte Carlo de forma que iremos variando cada una de nuestras *semillas*, tal y como nos dicte el algoritmo de Metrópolis. Resaltar que no tomaremos el resultado tras cada barrido, sino que dejaremos varios barrido entre resultado y resultado para evitar que las cadenas estén demasiado correlacionadas.

2.2.2. Conversión de un vector de $\mathbb{R}^{(20L)}$ a una secuencia de L aminoácidos.

En la sección 2.1.1 hemos visto cómo superar la dificultad de cambiar de formato las secuencias del archivo de learning para poder trabajar con ellas. El problema inverso, de pasar de una secuencia binaria de N= 20L bits a la secuencia de caracteres correspondiente es trivial, si el vector cumple con los vínculos relacionados a la imposibilidad de tener dos posiciones activas en el mismo bloque de 20 bits que representa una única posición, pues esto correspondería a tener dos aminoácidos en la misma posición de la secuencia. Sin embargo, los vectores generados de acuerdo a la distribución gaussiana no son binarios ni cumplen, en general, con esos vínculos. Se nos plantea pues el problema de partir de un vector real y obtener un vector binario que se corresponda a una secuencia admisible. Es decir, necesitamos que la cadena generada esté compuesta por 0's o 1's, pero nunca por valores intermedios. Para conseguir esto, definimos el parámetro *Límite*. Este parámetro funcionará como umbral, de forma que si la componente de nuestra secuencia generada es mayor que *Límite* la haremos 1 y si es menor, 0. El principal escollo es fijar *Límite*. Si es demasiado alto, muy pocas componentes llegarán a superarlo y tendremos demasiados gaps. Sin embargo, si lo fijamos con un valor demasiado bajo tendremos el caso contrario, lo superarán más componentes de las que deberían y no tendremos suficientes gaps.

[1] Podemos encontrar el artículo original en [5]
[2] Podemos ver una explicación detallada sobre los barridos de Monte Carlo en el Anexo 5.2

Para hallar el valor óptimo del parámetro *Límite*, generamos una base de datos artificial usando las librerías de Julia, haciendo después un barrido variando la variable *Límite*. Para cada valor de *Límite* transformaremos nuestra base de datos artificial de $\mathbb{R}^{(20L)}$ a una secuencia de L caracteres. Una vez hecho esto, compararemos cada una de nuestras cadenas con las secuencias de la base de datos obtenida experimentalmente. El valor de *Límite* para el que mayor número de coincidencias tengamos, será el que elijamos. Además distinguiremos a la hora de comparar entre posiciones de las cadenas que sean muy constantes en el archivo de secuencias experimentales de las que sean más variables. Si representamos la probabilidad de coincidencia promedio de todas las posiciones, obtenemos la Figura 4:

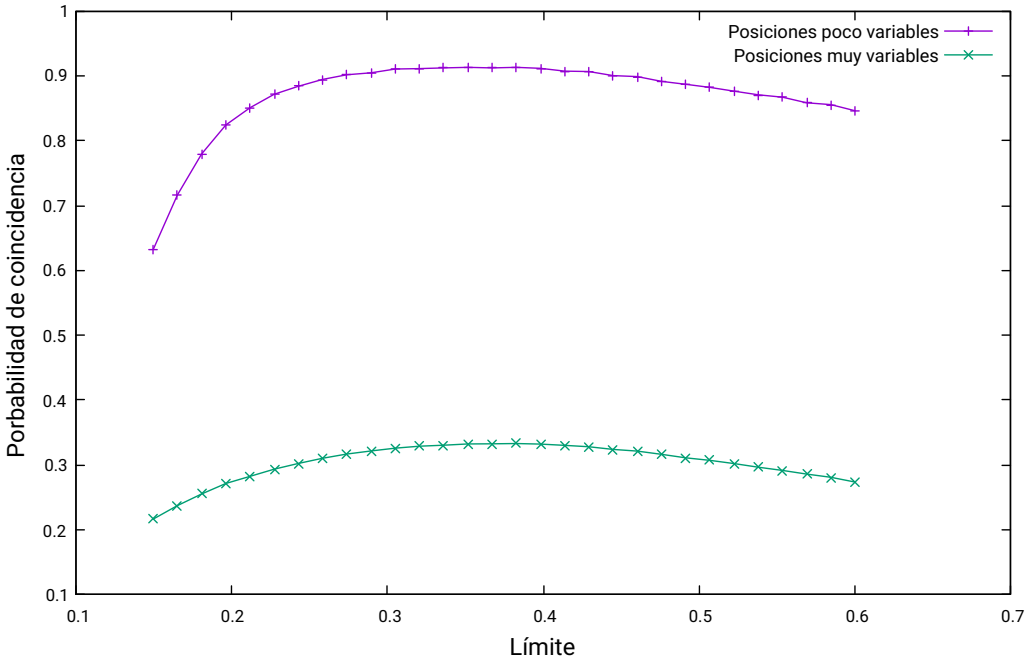


Figura 4: Obtención de Límite

Vemos en la Figura 4 que para ambos casos, independientemente de la variabilidad de una posición en el archivo 'de aprendizaje', el valor de *Límite* óptimo es 0.367. Podemos ver el código utilizado y un desarrollo más detallado del proceso en el Anexo 4: *Descodificación de cadenas: Obtención de Límite.*

2.3. Validación de los métodos para fijar los parámetros Omega y Lambda.

Una vez hayamos generado nuestras secuencias artificiales, validamos los procedimientos que nos permitirán hallar el valor óptimo de Ω y λ para esas cadenas.

En la determinación del valor óptimo de Ω a partir de nuestro archivo de cadenas artificiales vamos a considerar dos variables diferentes, pero relacionadas entre sí:
-La norma de Frobenius de la matriz de la covarianza pesada, siguiendo la ecuación 5.
-Como la opción anterior, pero en lugar de usar la matriz de covarianza usaremos la de correlación, cuya expresión será:

$$Corr_{ij} = \frac{\frac{1}{M_e}\sum_{m=1}^{M}(x_i^m - \bar{x}_i)(x_j^m - \bar{x}_j)w_m}{\frac{1}{M_e}\sqrt{\left(\sum_{m=1}^{M}(x_i^m - \overline{x}_i)w_m\right)\left(\sum_{m=1}^{M}(x_j^m - \overline{x}_j)w_m\right)}} \tag{12}$$

En el caso de querer encontrar el valor óptimo de λ, definiremos dos variables, d_1 y d_2, de la siguiente manera:

$$d_1 = \frac{|\Sigma_{cad} - \Sigma_{ref}|}{|\Sigma_{ref}|} \tag{13}$$

$$d_2 = \frac{|\mu_{cad} - \mu_{ref}|}{|\mu_{ref}|} \tag{14}$$

Siendo Σ_{cad} y μ_{cad} los parámetros obtenidos para la distribución gaussiana del archivo de cadenas que estemos estudiando y Σ_{ref} y μ_{ref} los obtenidos para el archivo de cadenas que usamos como learning. La idea es calcular d_1 y d_2 para un Ω obtenido previamente y ver si ambos parámetros se minimizan simultáneamente y en qué λ lo hacen.

2.4. Generación de secuencias con distintos hamiltonianos.

La generación de secuencias a partir de una distribución gaussiana es solo el primer paso en la generación de bases de datos de secuencias cada vez más complicados de aprender para el Modelo Gaussiano Multivariante. En concreto, es posible definir diferentes hamiltonianos, con cierta base experimental a partir del archivo de aprendizaje de [3], con o sin correlaciones. A continuación, trataremos una versión sencilla, sin correlaciones entre posiciones, que sin embargo nos ayudará en nuestros tests de los protocolos del modelo MGM. Además, resaltar que en este apartado continuaremos trabajando, esta vez en exclusiva, con el algoritmo de Metrópolis para obtener nuestras cadenas.

2.4.1. Hamiltonianos desacoplados dependientes de la posición

En la definición de esta familia de hamiltonianos, el primer paso será hallar la cadena moda de nuestro archivo 'de aprendizaje'. Es decir, formaremos una nueva secuencia compuesta por el aminoácido más abundante de cada columna. Tratamos así de conseguir que si nuestra cadena artificial y la secuencia moda no coinciden, la penalización energética no sólo dependa de la columna en la que son distintas, sino también de qué aminoácido incorrecto hemos puesto. Así, el hamiltoniano tendrá la siguiente forma:

$$H = \sum_{i=1}^{L} \epsilon_{i,\alpha}(1 - \delta_{\sigma_i \sigma_i^*}) \tag{15}$$

donde σ_i es el aminoácido que se encuentra en la posición i de nuestra cadena, σ_i^* es el aminoácido que se encuentra en la posición i de la cadena moda de nuestro archivo experimental y $\epsilon_{i,\alpha}$ es el coste energético si tenemos el aminoácido α en la posición i de nuestra cadena y no coincide con el aminoácido de la posición i de la cadena moda del archivo

experimental.
Esta es la expresión más general de un hamiltoniano que refleje los datos experimentales y no presente interacciones: las $\epsilon_{i,\alpha}$ tienen la función de campos externos, en principio independientes. Naturalmente, el modelo se puede simplificar, haciéndolo independiente del tipo de aminoácido (es decir $\epsilon_{i,\alpha} = \epsilon_i$), o incluso homogéneo en toda la secuencia ($\epsilon_{i,\alpha} = \epsilon$). Para fijar los valores $\epsilon_{i,\alpha}$ el tratamiento canónico. Siendo $p_{i,\alpha}$ probabilidad de que el aminoácido α aparezca en la posición i de nuestra cadena, planteamos:

$$f_{i,\alpha} = \frac{1}{Z} exp(-\beta\epsilon_{i,\alpha}) \tag{16}$$

Si despejamos de aquí $\epsilon_{i,\alpha}$, obtenemos:

$$\epsilon_{i,\alpha} = \frac{-1}{\beta}(ln(f_{i,\alpha}) - ln(Z)) \tag{17}$$

Observamos que los $\epsilon_{i,\alpha}$ dependen de la función de partición. Sin embargo, dado que vamos a usar un método de Monte Carlo, no necesitamos el valor absoluto de la energía de una cadena, sino la diferencia de energía entre las dos cadenas que queramos comparar, que no depende de la función de partición. Definimos así:

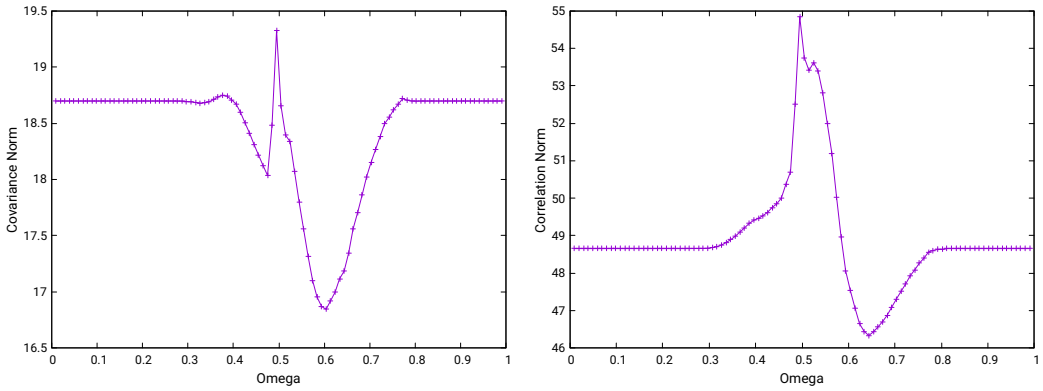
$$\epsilon_{i,\alpha} = \frac{1}{\beta} ln\left(\frac{1}{f_{i,\alpha}}\right) \tag{18}$$

Observamos que, si un aminoácido no aparece en una columna de nuestro archivo de cadenas obtenidas experimentalmente, pero sí que aparece en esa misma posición de nuestra cadena, nos costaría energía infinita. Cuando esto ocurra, fijaremos el $\epsilon_{i,\alpha}$ de forma arbitraria. Para fijarla, calcularemos cuanta energía nos cuesta colocar en nuestra cadena un aminoácido que unicamente aparezca una sola vez en toda la columna y fijamos un valor mucho mayor que ese, de forma tal que la energía asociada sea a efectos prácticos ïnfinita".

3. Resultados.

3.1. Validación de los programas con la base de datos 'de aprendizaje'.

Como ya hemos comentado anteriormente, usamos el parámetro Ω para moderar el peso de cada secuencia, pesando menos aquellas que se parezcan demasiado entre sí, tratando de evitar o reducir el sesgo existente. Lo primero que haremos será validar que, efectivamente, nuestro programa proporciona los mismos resultados que los obtenidos en [1] cuando lo aplicamos sobre el archivo de learning. Los resultados obtenidos para la covarianza y la correlación pueden verse en la Figura 5:



(a) Covarianza (b) Correlación

Figura 5: Validación de los programas

Observamos que para ambos criterios obtenemos un pico muy marcado justo en el 0.49, luego sí que somos capaces de recuperar el valor obtenido en [1] y ambos criterios indican el mismo valor. Además, llama la atención el hecho de que el primer mínimo relativo que aparece con el criterio de la covarianza desparece con el de la correlación. También nos fijamos en que el valor máximo de la correlación está mucho más alejado de su valor basal que el de la covarianza. Esto se debe a que en la correlación, Ecuación 12, interviene la desviación estándar de cada columna en el denominador, lo que también aumenta el valor basal. Se debe también a la desviación estándar el hecho de que en la covarianza el valor mínimo está más alejado del valor basal de lo que lo está el valor máximo y que, sin embargo, en la correlación esto se ha invertido, el valor mínimo es más cercano al basal que el valor máximo.

En el caso de λ no podemos usar el criterio que involucra a d_1 y d_2 porque es obvio que ambas se minimizarán para $\lambda = 0.1$, pues hemos usado estas mismas cadenas para calcular μ_{ref} y Σ_{ref}.

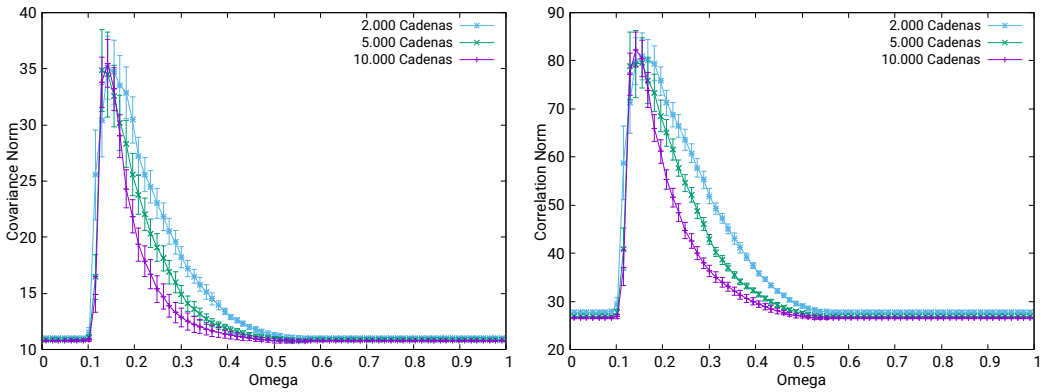
3.2. Validación del criterio para obtener Ω y λ.

A continuación, vamos a estudiar si el criterio propuesto en [1], que fija Ω como el valor que proporciona un máximo de la norma de Frobenius de la matriz de covarianza, representa la mejor elección. Esto también nos permitirá estimar cuál es el número mínimo de secuencias que debe tener la base de datos 'de aprendizaje' para proporcionar resultados fiables.Comenzaremos analizando una base de datos de secuencias independientes. Pasaremos luego a analizar el caso de una base de datos compuesta por bloques de secuencias repetidas, para finalmente tratar el caso más realista de un subconjunto de secuencias con mayor correlación entre sí.

3.2.1. Bases de datos artificiales de cadenas independientes.

En este primer caso, vamos a utilizar bases de datos generadas usando las librerías de Julia. Como hemos mencionado en la sección 2.2.1, al generar de esta forma las secuencias

serán estadísticamente independientes entre sí.
Durante la realización de este ejercicio, se percibe que la norma de Frobenius tanto de la covarianza como de la correlación es muy sensible a las cadenas de la base de datos. Ante esto se procede a presentar un resultado promedio obtenido a partir de 10 bases de datos distintas e independientes, para cada valor del número de cadenas que forman la base de datos. Además gracias a este procedimiento, podremos ver cómo varía la magnitud de la desviación estándar en cada serie de datos en función del número de secuencias que componen nuestra base de datos y estimar el número de cadenas mínimo necesario en una base de datos para que sea fiable. Así, siguiendo el procedimiento descrito en 2.3 obtenemos, los resultados de la Figura 6:

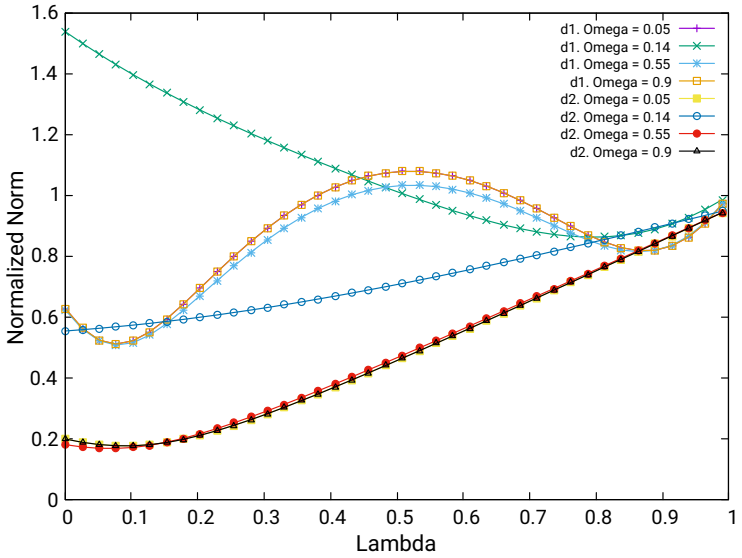


(a) Covarianza (b) Correlación

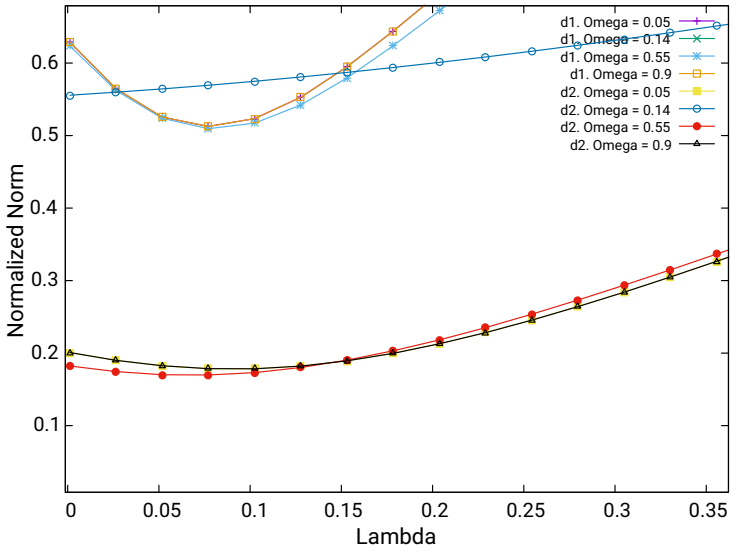
Figura 6: Barrido en Ω usando bases de datos de secuencias independientes

Observamos que tenemos un pico para ambos criterios, igual que en la Figura 5, pero este se corresponde para un valor de Ω mucho menor que en el otro caso. Esto parece ir en contra de lo intuitivo, pues recordemos que las cadenas dentro de una misma base de datos son independientes, por lo que parecería que hemos de pesar a todas las cadenas por igual, usando valores de Ω cercanos a 1. Sin embargo, cabe observar que, debido a que nuestra gaussiana de referencia está muy sesgada sobre los datos experimentales de la base de datos 'de aprendizaje' de [3] , las secuencias que genera son sí estadísticamente independientes, pero también tienen cierta similitud entre ellas. El hecho de que las normas de Frobenius de la covariancia y correlación comiencen a apartarse de la línea base en $\Omega = 0{,}1$ y continúen hasta $\Omega = 0{,}5$, implica que todas las secuencias tienen por lo menos $L\Omega \approx 30$ aminoácidos idénticos, y que algunas tengan alrededor de $L\Omega \approx 149$ en común (sin contar los gaps, en ambos casos); véase Eq. (2) y (3). Además observamos que, en el caso de la covarianza, el primer mínimo que aparecía en la Figura 5 aquí no aparece. También nos damos cuenta, si miramos los ficheros de datos de las bases de datos de las que hemos sacado los promedios representados en la Figura 6, de que para algunos casos sí que tenemos un mínimo, muy tenue, para un valor de Ω de 0.55, aproximadamente.
Ante estos resultados, tomamos una de las bases de 10.000 cadenas que hemos usado para hacer el promedio presentado en la Figura 6 y hacemos un barrido en λ calculado d_1 y d_2

para distintos valores de Ω. En concreto, tomaremos 0.05, 0.14, 0.55 y 0.9 por corresponderse con un valor de la norma de Frobenius de la covarianza basal, máximo y mínimo, respectivamente, para esa determinada base de datos escogida. Obtenemos la Figura 7:



(a) Barrido completo en λ



(b) Ampliación de (a)

Figura 7: d_1 y d_2 para 10000 secuencias independientes.

Observamos que el valor de Ω para el que peores resultados obtenemos es, precisamente, el que maximiza la norma de Frobenius de la covarianza y de la correlación. Vemos que, para $\Omega = 0.14$, d_1 se minimiza para un valor de $\lambda = 0.79$, mientras que d_2 lo hace para un valor de $\lambda = 0.001$. Es decir, que no sólo lo hacen para distintos valores de λ, sino que además esos valores están muy alejados. Observamos también que el motivo de que d_1 se minimice para un valor tan grande de Ω se debe a la desaparición del primer mínimo que

el resto de valores de Ω sí que presentan. Por último, destacar que para los dos valores extremos de Ω, que se corresponden con los valores basales de la norma de Frobenius de la covarianza y la correlación, obtenemos exactamente el mismo resultado. Hablaremos más en detalle sobre los valores extremos de Ω en la sección 3.2.2.
Si observamos ahora la Figura 7 b) vemos que el valor de Ω para el que mejores resultados obtenemos es 0.55, el cual minimizaba la norma de Frobenius de la covarianza y la correlación. No solo obtenemos los mejores resultados para $\Omega = 0.55$, sino que, además, d_1 y d_2 se minimizan de forma simultánea para $\lambda = 0.07$. Recordamos que valores pequeños de λ están asociados a un mayor peso de la componente fenomenológica de la Σ, frente a la contribución de la probabilidad previa, lo que es deseable para tener un modelo no trivial. Observamos también que la diferencia de la curva para $\Omega = 0.55$ no dista mucho de la dibujada para los dos valores extremos de Ω.

3.2.2. Bases de datos artificiales formadas por bloques de secuencias idénticas.

Ya hemos dicho anteriormente que las cadenas de nuestro archivo de learning no son estadísticamente independientes y que por eso íbamos a usar Ω para pesar menos las cadenas que sean demasiado similares entre ellas y evitar el sesgo existente. Hemos comentado también, en la sección 2.2.1, que para emular la dependencia estadística que existe en el archivo 'de aprendizaje', íbamos a generar *Nsem* secuencias independientes con las librerías de Julia y a aplicar el método de Monte Carlo sobre cada una de ellas. En esta sección construiremos un archivo constituido por la repetición de un bloque de *Nsem* secuencias independientes, que generaremos con las librerías de Julia, repetidas *Nrep* veces. Esto permite simplificar el cálculo de la matriz de covariancia para los valores límite de Ω: 0,1.

Caso en el que Ω tiende a 0.
Según las ecuaciones (3) y (4), con Ω tendiendo a 0, estaríamos considerando todas las cadenas, aunque no tengan ningún aminoácido en común. Esto implica que pesamos a todas las cadenas por igual. En concreto, siendo M el número total de cadenas que tenemos:

$$w_m = \frac{1}{M} \quad \forall m = (1, ..., M) \tag{19}$$

Por lo que, si calculamos M_e con la ecuación (5) obtenemos:

$$M_e = \sum_{m=1}^{M} w_m = \sum_{m=1}^{M} \frac{1}{M} = 1 \tag{20}$$

Nos fijamos también en que podemos descomponer M de forma que:

$$M = N_{sem} N_{rep} \tag{21}$$

Aplicando esto a la ecuación (7), obtenemos:

$$C_{ij} = \frac{1}{N_{sem} N_{rep}} \sum_{m=1}^{M} (x_i^m - \bar{x}_i)(x_j^m - \bar{x}_j) \tag{22}$$

Ahora vamos a aplicar que en realidad no tenemos N cadenas independientes, sino Nsem pero repetidas Nrep veces. Esto quiere decir que no haría falta hacer el sumatorio de la ecuación anterior desde 1 hasta M, sino que con hacerlo hasta Nsem bastaría, ya que luego comenzaríamos a repetir términos. En concreto, si hacemos el sumatorio hasta Nsem lo obtendremos Nrep veces repetido. Así:

$$C_{ij} = \frac{1}{N_{sem}} \sum_{m=1}^{N_{sem}} (x_i^m - \bar{x}_i)(x_j^m - \bar{x}_j) \tag{23}$$

Caso en el que Ω tiende a 1.

Seguimos trabajando con un archivo con la misma estructura que en el caso anterior, un bloque de Nsem repetido Nrep. Por ello, continúa vigente el argumento de que el sumatorio de la ecuación (7) no tiene que llagar hasta M, sino que con llegar hasta Nsem y multiplicar por Nrep, bastaría. Así, planteamos:

$$C_{ij} = \frac{N_{rep}}{M_e} \sum_{m=1}^{N_{sem}} (x_i^m - \bar{x}_i)(x_j^m - \bar{x}_j) w_m \tag{24}$$

Dado que Ω tiende a 1,según la Ecuación 2, una secuencia solo es igual a si misma, así que $w_m = 1$ para todas las secuencias, por lo que en este caso tendremos:

$$M_e = \sum_{m=1}^{M} w_m = \sum_{m=1}^{M} 1 = M = N_{sem} N_{rep} \tag{25}$$

Al sustituirlo en la ecuación (27), obtenemos:

$$C_{ij} == \frac{1}{N_{sem}} \sum_{m=1}^{N_{sem}} (x_i^m - \bar{x}_i)(x_j^m - \bar{x}_j) \tag{26}$$

Si comparamos las ecuaciones (28) y (31), observamos que son exactamente iguales. Esto arroja un resultado interesante, y es que el valor de la norma de Frobenius de la covarianza vale lo mismo para Ω muy pequeña, tendiendo a 0 y para Ω muy grande, tendiendo a 1. Además, C_{ij} depende también del número de semillas, aunque su efecto en las curvas no es fácil de predecir.

Lo siguiente que haremos es verificar este resultado. Comprobaremos dos cosas:

1.- Que el valor de C_{ij} no depende de Nrep. Tomaremos un bloque de Nsem y construiremos distintos archivos con distintos Nrep cada uno. Luego calcularemos Ω para cada archivo y los presentaremos todos en la misma gráfica.

2.- Cómo varia el valor de C_{ij} cuando aumentamos Nsem. Repetiremos el proceso del punto anterior, pero fijando esta vez Nrep y variando Nsem en los distintos archivos.

En la Figura 8, vemos los resultados obtenidos por una muestra de 50 semillas, y diferentes repeticiones exactas. En Figura 9 se ven los resultados obtenidos con muestras diferentes de diferentes números de semillas, y el mismo número de repeticiones.

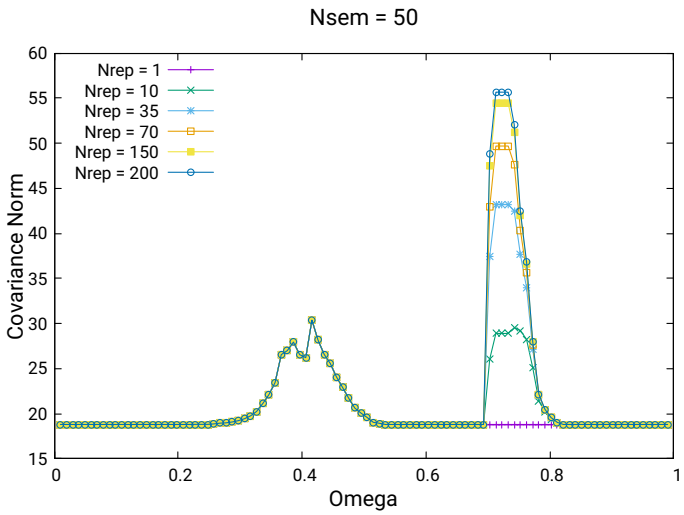


Figura 8: Fijamos Nsem y variamos Nrep.

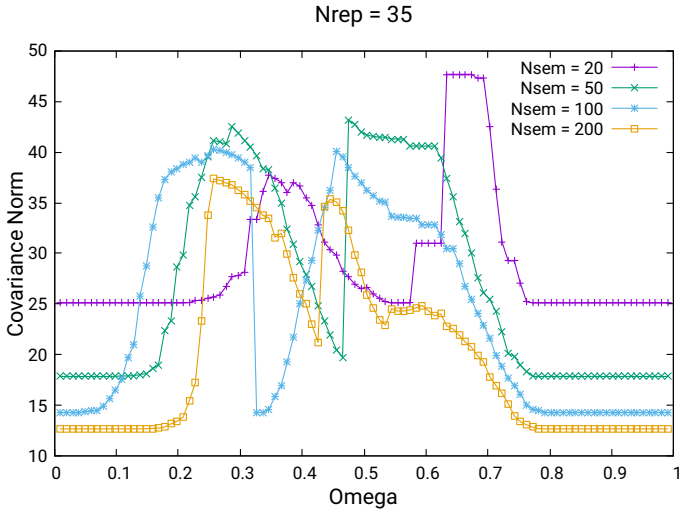


Figura 9: Fijamos Nrep y variamos Nsem.

Con estas dos gráficas hemos comprobado de forma clara que para los dos valores extremos de Ω, el valor de la norma de Frobenius de la covarianza coincide y que, efectivamente, al aumentar Nsem, reducimos el valor de la Norma de Frobenius de la covarianza en los extremos. Observamos también que la curva dibujada en la Figura 8 y la dibujada en la Figura 9 no son iguales, a pesar de estar ambas generadas con Nsem $= 50$ y Nrep $= 35$. Esto se debe a que la norma de Frobenius de la covarianza es muy sensible a las cadenas que conforman la base de datos y varían de muestra a muestra. Si quisiésemos obtener resultados fiables, tendríamos que imitar el proceso realizado en la sección 3.2.1. Dado que únicamente queríamos comprobar que el valor basal no depende de Nrep y que al aumentar Nsem se reduce, no será necesario promediar. Sin embargo, es relevante notar cómo en todos casos aparecen dos máximos en Figura 9, de acuerdo también a lo que se ve en Figura 8.

Si miramos de nuevo la Figura 8, nos llama la atención el segundo pico que aparece, el que se posiciona más a la derecha. Deducimos que se trata de que la covarianza detecta que tenemos cadenas repetidas. Es decir, tenemos un primer pico que nos indica cómo de relacionadas están las cadenas dentro de un mismo bloque y un segundo pico que nos indica que tenemos cadenas muy iguales, es decir, las repetidas. Vemos en la primera gráfica que este se hace más pronunciado cuantas más veces repitamos nuestro bloque de cadenas y que se anula si no lo tenemos repetido. Recordamos que un valor de Ω pequeño implica que consideramos muchas secuencias como homólogas, y las pesamos menos, mientras que Ω cercano a 1 implica que todas las secuencias se consideran distintas y, por tanto las pesamos más.

Si observamos la Figura 9, vemos que al comparar la diferencia entre los valores basales de la norma de Frobenius de la covarianza para dos valores de Nsem consecutivos, esta se reduce al aumentar el valor de los Nsem que estamos comparando. Podemos verlo también en la Figura 6, donde hemos representado bloques de 10.000, 5.000 y 2.000 secuencias independientes, que se comportarían como las semillas con las que trabajamos en esta sección. Ante esto procedemos a representar el valor basal de la norma de Frobenius de la covarianza frente al número de semillas, obtenemos la Figura 10 :

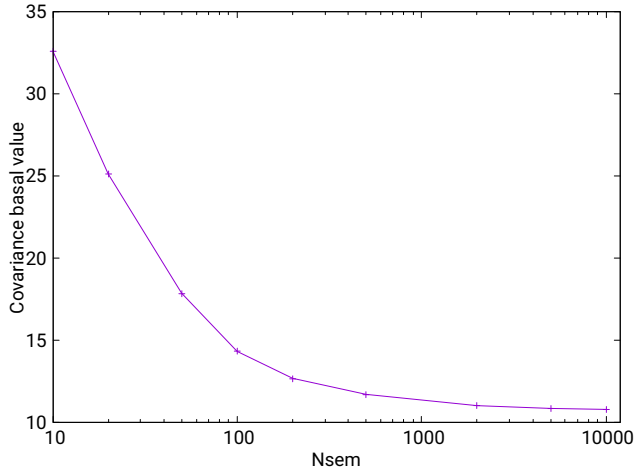


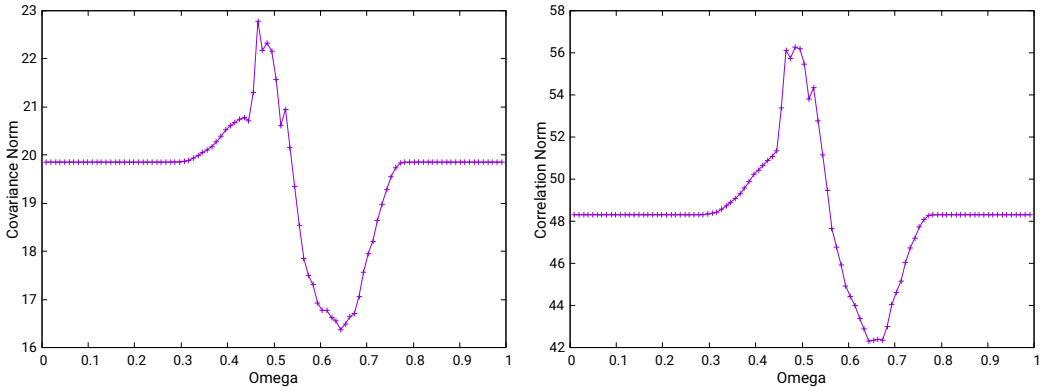
Figura 10: Valores basales de la norma de Frobenius de la covarianza frente a Nsem.

Vemos que, efectivamente, el valor basal tiene un comportamiento asintótico, se reduce rápidamente cuando aumentamos Nsem hasta llegar a un valor Nsem = 500, donde comienza a saturarse, hasta llegar a quedar prácticamente estable cuando llegamos a Nsem = 5000.Esto también sugiere que el número de secuencias M en la base de datos no debería ser inferior a 5000, para poder confiar en los resultados.

3.2.3. Bases de datos artificiales de cadenas correlacionadas.

En este caso vamos a utilizar las librerías de Julia para generar cadenas, las llamadas semillas, sobre las que luego aplicaremos Monte Carlo para obtener así nuestra base de datos, tal y como detallamos en la sección 2.2.1. El tener que usar Monte Carlo limitará

mucho la extensión máxima de las bases de datos, pues el coste computacional para generar las secuencias es ahora mucho mayor que en las secciones 3.1 y 3.2.2, aunque nos dará un resultado lo suficientemente estable estadísticamente como para no tener que realizar varias veces el proceso completo y mostrar el promedio. Para rebajar ese coste computacional y dado que no queremos hacer una exploración ergódica del espacio de secuencias, limitaremos los aminoácidos posibles en cada posición. Esto lo haremos a partir de nuestra base de datos 'de aprendizaje' experimental tomada de [3], tomaremos una de sus columnas, veremos qué aminoácidos aparecen en ella y esos serán los únicos posibles en esa posición de nuestra cadena. Además, limitaremos el número de cambios que puede sufrir una cadena en cada barrido de Monte Carlo a un 20 % de su longitud. Este proceso lo fijamos así porque en nuestra base de datos 'de aprendizaje' tenemos columnas que son muy estables, lo que se traduce en una aceptancia de cambios muy pequeña en esas columnas, por lo que deja de tener sentido tratar de cambiarlas. Además, el hecho de que en un barrido de Monte Carlo recorramos únicamente el 20 % de las posiciones de nuestra cadena, hace más eficiente nuestro programa en términos de tiempo de computación.
Obtener de esta forma la base de datos implica tener un grado de libertad adicional; el número de semillas del que partimos. De acuerdo a los resultados obtenidos en la sección 3.2.2 sabemos que el valor basal depende del número de semillas que tengamos. A pesar de que en este caso no tengamos bloques de cadenas idénticas, al estar correlacionadas esa dependencia seguirá existiendo. Así, tratamos de, al generar 1000 secuencias, obtener el valor basal de la Figura (5). Esto lo conseguiremos con 175 semillas. Así, generamos 1000 secuencias partiendo de 175 semillas y hacemos un barrido en Ω, obteniendo las curvas de la Figura (11):

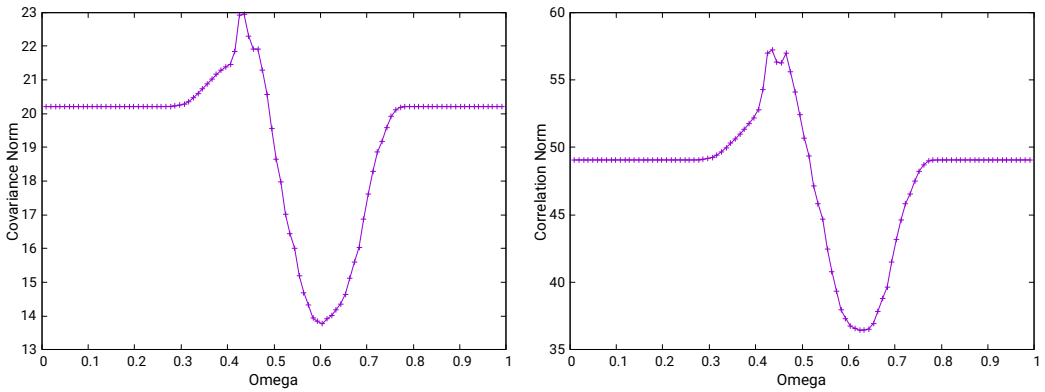


(a) Covarianza (b) Correlación

Figura 11: Base de datos de 1000 cadenas a partir de 175 semillas

Destacar que, aunque no tan acusadamente como en la sección 3.2.1, la covarianza sigue dependiendo de las secuencias que conforman la base de datos. Sin embargo, tras generar distintas bases de datos de 1000 cadenas a partir de 175 semillas, observamos que los valores de Ω para los que obtenemos el máximo y el mínimo de la norma de Frobenius tanto de la covarianza como de la correlación se mantienen aproximadamente constantes.

Si observamos la Figura 11, puede parecer que logramos reproducir la forma de la Figura 5 tanto para la covarianza como para la correlación, exceptuando el primer mínimo que presentaba la covarianza en la Figura 5. Sin embargo, en esta última vemos que tanto el valor máximo en la covarianza como el mínimo en la correlación están más cercanos al valor basal de lo que lo están en la Figura 11. Ante esta situación vamos a generar una nueva base de datos más extensa, pero manteniendo el ratio semillas/secuencias, para tener más fiabilidad estadística a la hora de validar los protocolos de Ω y λ. Así, generamos una base de 5000 secuencias partiendo de 870 semillas, barremos en Ω y obtenemos la covarianza y la correlación de la Figura 12.



(a) Covarianza (b) Correlación

Figura 12: Base de datos de 5000 cadenas a partir de 875 semillas

En esta figura se aprecia que los resultados mantienen la misma forma que cuando teníamos 1000 cadenas, Figura 11, aunque ha aumentado el valor basal y el valor mínimo de ambos criterios se ha alejado más de este. El siguiente paso será, usando esta base de datos de 10000 cadenas, hallar el valor de d_1 y d_2 barriendo en λ para distintos valores de Ω, como se hizo en la sección 3.2.1. En concreto, tomaremos como valores de Ω: dos extremos, 0.1 y 0.9, y los valores aproximados donde presentan los máximos y mínimos la covarianza y la correlación, 0.44 y 0.61. Los resultados pueden verse en la Figura 13

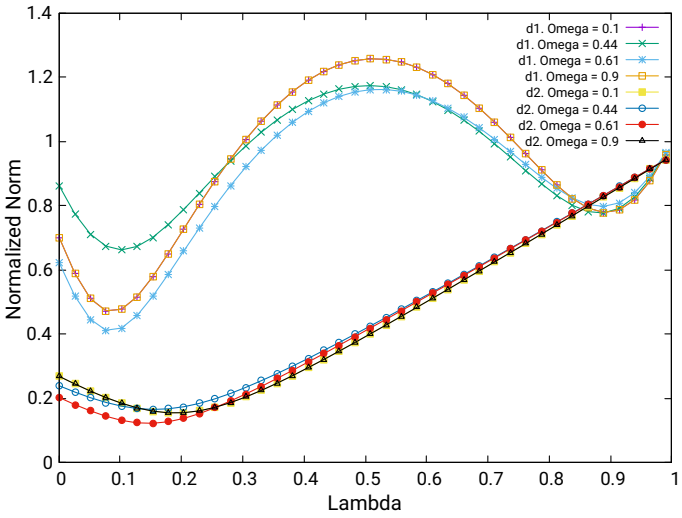
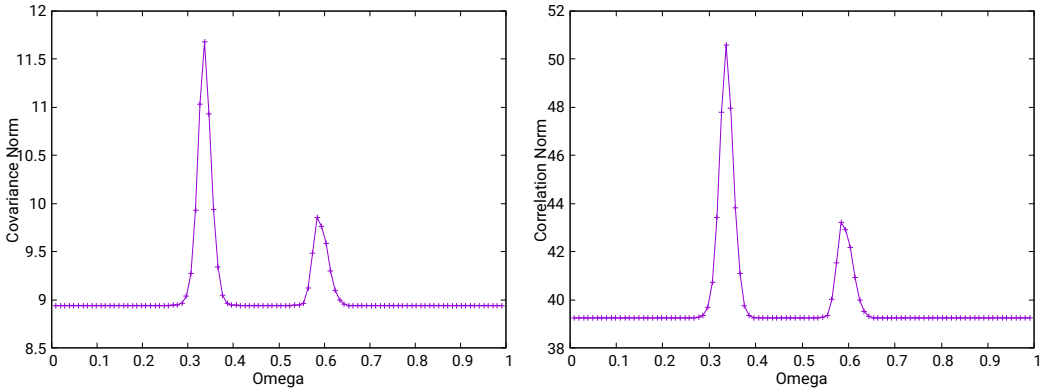


Figura 13: d_1 y d_2 para 5000 secuencias correlacionadas.

, donde se observan varias cosas: es interesante el hecho de que el mejor resultado no lo obtenemos con el valor de Ω que maximiza la norma de Frobenius de la covarianza y de la correlación, sino el que la minimiza. Además, d_1 y d_2 no se minimizan de forma simultánea En concreto, para $\Omega = 0.62$, d_1 lo hace para $\lambda = 0{,}8$ y d_2 para $\lambda = 0.15$. Por último resaltar que hemos obtenido exactamente el mismo resultado para los dos valores extremos de Ω, tal y como predijimos en la sección 3.2.2.

3.2.4. Bases de datos artificiales de cadenas con frecuencias experimentales.

En esta sección vamos a utilizar el hamiltoniano presentado en 2.4.1. Sin embargo, vamos a realizar una modificación del método que usamos para generar las secuencias,siguiendo el protocolo 3.2.3. Recordemos que, en esa sección, generábamos todas nuestras cadenas a partir de secuencias semilla. En este caso, no partiremos de una cadena con una distribución plana, para luego termalizarla hasta llevarla a una distribución determinada por la frecuencia de aparición de cada aminoácido en cada columna en nuestro archivo 'de aprendizaje', y luego a partir de esa cadena termalizada hallar el resto haciendo barridos de Monte Carlo. Lo que haremos será llevar varias cadenas a la distribución determinada por la frecuencia de aparición de cada aminoácido en cada columna en nuestro archivo 'de aprendizaje' y serán esas secuencias termalizadas las que actuarán como semillas. Además, para lograr una mayor similitud con lo expuesto en la sección 3.2.3 limitaremos en este caso también el cambio máximo de una secuencia en un barrido de Monte Carlo a un 20 % de su longitud. Dado que usando este hamiltoniano el coste computacional de generar cadenas es pequeño, podemos generar una base de datos de 10.000 secuencias y luego barrer en Ω. Obtenemos las curvas de la Figura 14:



(a) Covarianza (b) Correlación

Figura 14: Barrido en Ω usando una base de datos obtenida con el hamiltoniano definido en la sección 3.2.4, tomando como Nsem = 1750.

Observamos que tenemos dos picos en ambas figuras, pero ningún mínimo, hecho similar a la situación que nos encontrábamos en la Figura (9). Observamos también que, a pesar de haber utilizado 1750 semillas para generar la bases de datos, manteniendo así el ratio secuencias/semillas que proponíamos en la sección 3.2.3, el valor basal en el caso de ambos criterios se ha reducido de forma notable. El hecho de no tener mínimo nos deja sin un candidato claro para ser el valor óptimo de Ω.

4. Conclusiones.

En este trabajo nos hemos centrado en la validación del modelo gaussiano propuesto en [1] y [2]. El trabajo realizado se puede concretar en los siguientes puntos:
Después de una actualización y optimización de los códigos de [1], hemos desarrollado diferentes códigos en Julia para generar los distintos tipos de bases de datos de secuencias, así como para generar barridos en los parámetros Ω y λ del modelo, buscando los valores óptimos según el criterio de distancia con la distribución de referencia. El primer paso ha sido tratar de aprender con el modelo MGM una base de datos generada también con un modelo gaussiano de referencia. Para este fin, hemos tenido que encontrar un criterio fiable para reconstruir secuencias de aminoácidos a partir de vectores de números reales. Generando bases de datos de secuencias estadísticamente independientes, y otros con secuencias repetidas o correlacionadas, hemos podido valorar la validez de los protocolos establecidos en [1] para fijar los parámetros del modelo, y especialmente Ω. Dados los resultados presentados, podemos extraer las siguientes conclusiones:

- Hemos comprobado la existencia de un valor óptimo de *Límite* para transformar las secuencias de un vector de $\mathbb{R}^{(20L)}$ a una secuencia de L aminoácidos.

- Hemos comprobado que la norma de la covariancia fenomenológica tiene el mismo valor para Ω = 0 y Ω = 1, al barrer en Omega, se aleja de esta línea base atravesando un máximo, que corresponde a las secuencias más diferentes. A continuación, pueden darse un mínimo o un máximo, según la naturaleza de las correlaciones entre las

secuencias de la base de datos. No hemos podido determinar las causas que determinan la presencia del mínimo o del máximo.

- A partir de los barridos en λ para ciertos valores de Ω hemos encontrado que el valor óptimo de Ω es aquel que minimiza las normas de Frobenius de la covarianza y la correlación, y no aquel que la maximiza como se asumía en [1].
- d_1 y d_2 no se minimizan siempre de forma simultánea para un determinado valor de λ.
- Para valores extremos de Ω, tanto la norma de Frobenius de la covarianza y de la correlación y las variables d_1 y d_2 mantienen su valor.
- Los valores basales de la norma de Frobenius de la covarianza y de la correlación presentan un carácter asintótico frente al número de semillas que usamos para generar la base de datos.
- Hemos comprobado también la alta sensibilidad de la covarianza a las cadenas que forman la base de datos y la correlación que exista entre ellas.
- Hemos obtenido que para poder obtener resultados fiables de una base de datos, esta no debe contener menos de 5000 secuencias.

Sin embargo, este modelo admite un mayor desarrollo del que hemos podido llevar a cabo. Algunos puntos a desarrollar en el futuro son:

- Optimización de la función encargada de calcular el peso de cada una de las cadenas para poder trabajar con bases de datos de una mayor dimensión.
- Implementación de un nuevo hamiltoniano que tenga en cuenta las interacciones de largo alcance.
- Deducción e implementación de un nuevo protocolo para obtener la dimensión mínima de una base de datos que de resultados fiables.

Referencias

[1] Clavero-Álvarez, A., Di Mambro, T., Perez-Gaviro, S. et al. Humanization of Antibodies using a Statistical Inference Approach. Sci Rep 8, 14820. https://doi.org/10.1038/s41598-018-32986-y (2018)

[2] Baldassi C, Zamparo M, Feinauer C, Procaccini A, Zecchina R, Weigt M, et al. (2014) Fast and Accurate Multivariate Gaussian Modeling of Protein Families: Predicting Residue Contacts and Protein-Interaction Partners. PLoS ONE 9(3): e92721. https://doi.org/10.1371/journal.pone.0092721

[3] Pilar Calvo Príncipe, Refinement of a statistical model for antibody humanization"(September 2019)

[4] M. C. Tays Hernández García,Estudio a nivel molecular de la polirreactividad idiotípica y el reconocimiento de linfocitos B por el anticuerpo B7Y33,(2011)

[5] Nichola Metrópolis, Arianna W.Rosenbluth, Marshall M.Rosenbluth and Agusta H.Teller, Equation of a State Calculations by Fast Computing Machines (June 1953)

[6] *https* : *//www.investigacionyciencia.es/blogs/medicina − y − biologia/*28*/posts/ste − es − el − rbol − de − tu − vida* − 10631

5. Apéndices.

5.1. Distribuciones.jl

En este anexo vamos a mostrar y desgranar el código que usamos para generar las cadenas con una distribución gaussiana, usando precisamente la librería de Julia de *Distribuciones.jl*.

Lo primero que tenemos que hacer para usar este método es generar el objeto *distribución*. Concretamente, lo hacemos con la instrucción:

```
Dist = MvNormal(media,cov)#Generamos el objeto distribucion.
```

Lo siguiente será crear una cadena de la longitud que nosotros queramos, en nuestro caso, 5960. Para ello, primero generamos una cadena de tipo Float64 de longitud 5960 llena de $undef$. Cuando le pasemos esta cadena a nuestro objeto *distribución*, este nos la devolverá ya modificada. Podemos ver como hacemos esto en las siguientes líneas de código:

```
Cadena=Array{Float64}(undef,5960) # Generamos la cadena 5960.
rand!(Dist,Cadena) # Llamamos al objeto distribucion.
```

Con esto ya tendríamos nuestra cadena de Float64 con distribución gaussiana. Siguiendo los pasos especificados en el apartado de descodificación de cadenas, el siguiente paso sería convertir esta cadena en una cadena de 1's y 0's para luego volver a transcribirla a una cadena de tipo Char. Este proceso lo hacemos de la siguiente manera:

```
global Limite = 0.367 #Fijamos el valor de Limite
function CritOnes(Cadena)
#Pasamos de numeros entre 0 y 1 a 0 o 1
for i in 1:length(Cadena)
Cadena[i]>Limite ? Cadena[i]=1.0 : Cadena[i]=0.0
#Si la componente de nuestra cadena es mayor que
```

```
#Limite la igulamos a 1 y si no la hacemos 0.
end
return Cadena
end
CadenaI=Int.(CritOnes(Cadena)) #Truncamos las componentes de
#nuestra cadena para que sean Int y no Float
```

Con esto ya tenemos nuestra cadena de 0's y 1's distribuida de forma gaussiana. Por último, sólo quedaría usar la función binarytoascii() tomada de Ref[4] para pasarla a Char. Con estos pasos ya tenemos nuestra cadena terminada.

5.2. Algoritmo de Metrópolis: Funcionamiento e Implementación

Este algoritmo nos permite conseguir una cadena que siga una distribución predeterminada partiendo de otra cadena con una distribución totalmente distinta. En nuestro caso partiremos de una cadena con distribución plana y llegaremos hasta una distribuida de forma normal.

Así, lo primero será generar dicha cadena con distribución plana. Lo hacemos de la siguiente manera:

```
global LongC = 298
Cadcand=rand(aminosdict, LongC) #Generamos una cadena random de
    longitud C
```

Ahora haremos lo que se conoce como un barrido. Hacer un barrido consiste en visitar en un orden aleatorio todas las posiciones de la cadena, dándoles así la oportunidad de cambiar de valor a cada una de ellas. En concreto el orden en el que actuaremos será el siguiente:
1-. Seleccionamos una posición de la cadena.
2-. Seleccionamos el aminoácido por el que vamos a tratar de cambiar la componente seleccionada.
3-. Creamos una nueva cadena igual a la anterior salvo en la componente seleccionada, que ponemos el aminoácido elegido en el paso 2.
4-. Calculamos el cociente entre la probabilidad de ambas cadenas, cociente que denominaremos C.
5-. Si la cadena del paso 3 es más probable que la inicial aceptamos el cambio y si no lo es aceptamos con probabilidad C.
Una vez hayamos completado estos 5 pasos para todas las componentes de la cadena, habremos hecho un barrido. Destacar que haremos estos barridos en el espacio de N reales, no en el de L caracteres, debido a que ambos son equivalentes y sale más rentable en cuanto a tiempo de ejecución del programa y comodidad hacerlo en el de N reales.

Necesitaremos realizar un número determinado de barridos para que nuestra cadena llegue a una distribución normal, si se quiere más información sobre cuantos barridos realizar antes de que nuestra cadena alcance la distribución normal véase el Anexo 4:*Algoritmo de Metrópolis: Termalización*. Una vez nuestra cadena siga una distribución gaussiana guardaremos la cadena como resultado cada cierto número de barridos.

```
OrdenCamb=randperm!(Vector{Int}(undef,LongC))#Vector que
#contiene el orden en el que vamos a visitar las componentes de la
    cadena.
PosCamb=rand(aminosdict, LongC)#Aminoacidos con los que vamos a
    intentar el cambio.
for h in 1:LongC
#Recorremos toda la cadena
CadAux[OrdenCamb[h]]=PosCamb[h]
#Variamos la cadena en una componente.
C=exp(0.5*(Expo))
#Calculamos C
if C>rand()

Cadcand=copy(CadAux)
#Aceptamos el cambio
else
CadAux=copy(Cadcand)
#No aceptamos el cambio
end
end
```

Esta ha sido una explicación general del algoritmo. Sin embargo, podemos concretar un poco más en nuestro problema. Si nos fijamos bien, cuando hablamos de calcular el cociente de probabilidades, tendríamos que calcular C haciendo la exponencial de la diferencia de las normas cuadráticas de Σ_{ref} con nuestra cadena principal y nuestra cadena modificada. Eso implica que hemos de hacer, al menos, una norma cuadrática de una matriz 5960 x 5960 y dos vectores de longitud 5960. Sin embargo, como ya se advierte justo después de enunciar esa expresión, en nuestro caso únicamente variamos, como mucho, dos elementos de la cadena binaria al cambiar un aminoácido por otro. Es decir, estamos o moviendo un 1 de posición (si ni el nuevo aminoácido ni el que vamos a cambiar son gaps) o creando o anulando un uno (si introducimos un gap o lo cambiamos por un aminoácido) o dejando la cadena invariante (si tratamos de cambiar un gap por un gap). Luego, no hace falta calcular la norma cuadrática entera, basta con saber cómo se transforma un norma cuadrática cuando cambiamos 2 elementos.

La transformación de una norma cuadrática cuando cambiamos 2 elementos de los vectores puede ser un poco complicada, así que vamos a hacerlo en dos pasos. Es decir, vamos a seguir los siguientes pasos:
1.-Vamos a crear una cadena binaria intermedia. Es decir, vamos a crear una cadena igual que nuestra cadena inicial pero vamos a anular el grupo de 20 posiciones que representa la

posición que vamos a cambiar. De esta manera, la diferencia entre nuestra cadena inicial y esta cadena que llamaremos *Cadena Intermedia* es de, como mucho, una posición, igual que entre nuestra cadena inicial al cambiar un aminoácido y esta Cadena intermedia.
2.-Calculamos las diferencias entre las normas cuadráticas de la cadena intermedia con la inicial sin modificar y la inicial modificada. 3.-Sumamos ambas diferencias.
4.-Nuestro factor C será el resultado de hacer la exponencial del resultado del punto 3 multiplicado por un medio.

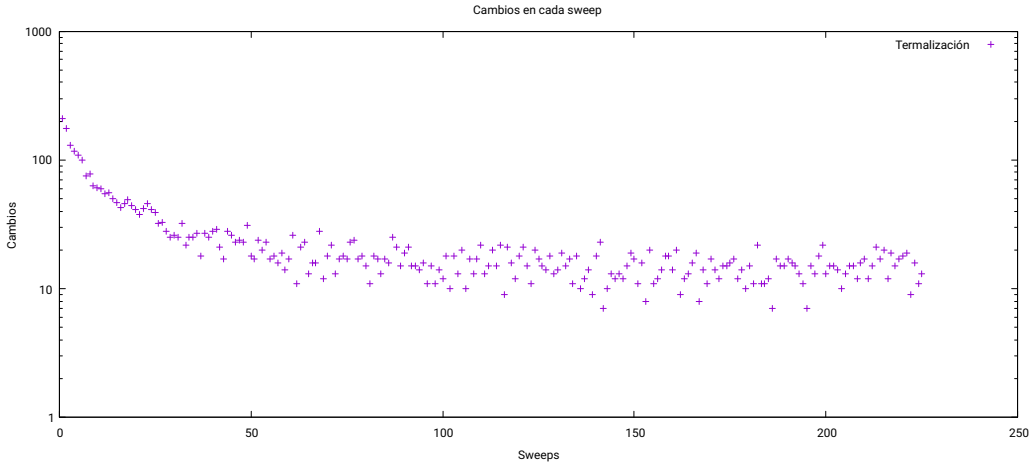
Veamos cómo introducimos esto en nuestro programa:

```
function Prepara(Cad,med)
#Pasa una cadena a binario y le resta el vector media.
CadB=asciitobinary(Cad)
vect=CadB-med
return vect,CadB
end

function Contribucion(invsig,vect, Posv)
#Calcula la contribucion de una componente del vector a la norma cuadratica.
T=-vect[Posv]*(dot(invsig[Posv,:],vect)+dot(invsig[:,Posv],vect)-invsig[Posv,Posv]*vect[Posv])
return T
end

function Delta(invsig,med,Cadcand,CadAux,PosC,AminC)
#Calcula el factor detallado en el punto 3.
vect,CadCB=Prepara(Cadcand,med)
vectAux,CadAB=Prepara(CadAux,med)
aux=copy(vect)
#aux sera nuestra cadena intermedia.
if Cadcand[PosC] != aminosdictD[21]
PosOld=(PosC-1)*20+findfirst(isequal(1),CadCB[((PosC-1)*20+1):PosC*20])
#Buscamos la poscion del 1 en el bloque de 20 que representa el aminoacido
aux[PosOld]=vectAux[PosOld]
#Terminamos de crear aux
T1=-Contribucion(invsig,vect,PosOld)+Contribucion(invsig,aux,PosOld)
#T1 es la diferencia entre la norma cuadratica usando la cadena intermedia y la inicial.
else
T1=0
#Si la cadena inicial tiene un gap en esta posicion no hay diferencia entre ella y la cadena intermedia, asi que este factor es 0.
end

if AminC != 21
Posv=(PosC-1)*20+findfirst(isequal(1),CadAB[((PosC-1)*20+1):PosC
```

```
    *20])
#Buscamos la poscion del 1 en el bloque de 20 que representa el
    aminoacido
T2=-Contribucion(invsig,aux,Posv)+Contribucion(invsig,vectAux,Posv)
#T2 es la diferencia entre la norma cuadratica usando la cadena
    intermedia y la inicial modificada.
else
T2=0
#Si el aminoacido por el que vamos a cambiar no hay diferencia entre
     ella y la cadena intermedia, asi que este factor es 0.
end

return (T1+T2) #Devolvemos el factor referido al punto 3.
end
```

5.3. Algoritmo de Metrópolis: Termalización

En esta sección veremos cuántos *barridos* tiene que hacer nuestro sistema antes de que podamos empezar a tomar resultados.
Para ello lo que haremos será poner nuestro programa a trabajar e iremos guardando cuantos cambios aceptamos cada *barrido*. Al principio veremos que ese número es muy grande ya que la distribución plana con la que hemos generado la primera cadena se aleja mucho de la distribución normal que queremos lograr, sin embargo ese número se irá estabilizando hasta mantenerse, no constante porque habrá fluctuaciones, pero si estable.
Para que no haya dudas, representamos el número de cambios realizados en función del *barrido* en el que nos encontramos, aunque lo hacemos poniendo escala logarítmica en el eje y para ver cuando de verdad se estabiliza el sistema de una forma más clara.
Así:



Vemos que con dejar 100 *barridos* como termalización será suficiente.

5.4. Descodificación de cadenas: fijar Límite

Vamos a hallar un valor para el parámetro límite del que hablábamos en la sección *Descodificación de cadenas*. El método que llevaremos a cabo consiste en barrer los posibles valores de Límite, de forma que, para cada iteración, genera 5000 cadenas, las pasa a tipo *Char* con el método desarrollado en la sección 2.2.2 y luego las compara con la base de datos columna a columna de la siguiente manera:

- Si la columna en cuestión tiene muy poca variabilidad en la base de datos, es decir, está siempre formada por el mismo aminoácido, simplemente comprueba que el aminoácido de nuestra cadena sea el mismo que aminoácido moda de esa columna en nuestra base de datos. Podemos observarlo en el código del programa en la variable *SumaE*.

- Si por el contrario estamos ante una columna muy variable, iremos comparando el aminoácido de nuestra cadena con todos los de la base de datos en esa posición, cadena a cadena.Este caso lo veremos plasmado en la variable *SumaI* cuando estudiemos el código.

Para distinguir en cuál de los dos casos estamos, diremos que una columna es poco variable cuando el aminoácido más frecuente aparezca más de un 80 % en la columna en cuestión.
Por último, presentamos el código del programa:

```
function Hallacolumnest()
#Nos devuelve un array (Estables) que contiene true si la cadena que corresponde a ese indice es estable o false en caso contrario y otro array que contiene las modas de cada una de esas columnas estables.
Moda=Array{Char}(undef,Lcad)
Estables=Array{Bool}(undef,Lcad)
#Definimos los dos arrays donde almacenaremos los indices de las columnas estables y sus modas.
for i in 1:Lcad
#Barremos en todas las columnas
Moda[i]=StatsBase.mode(seqs[:,i])
#Calculamos la moda de TODAS las columnas
porc=length(findall(isequal(Moda[i]),seqs[:,i]))/Dim[1]
#Calculamos que porcentaje de toda la columna representa la moda
porc > LimEst ? Estables[i]=true : Estables[i]=false
#Si el porcentaje que representa la moda es mayor que uno estipulado previamente (LimEst) la consideramos estable.
end
return Estables,Moda
end
const Limrange = range(0.15,stop=0.65,length=100)
#Vamos a barrer desde 0.15 hasta 0.65 haciendo 100 pasos intermedios.
for Limite in Limrange
for i in 1:Ncad
rand!(Dist,Cadena)
```

```
CadenaI=Int.(CritOnes(Cadena,Limite))
CadenaA=Binarytoasciig(CadenaI)
#Con estas 3 ultimas lineas hemos generado una cadena.(Como se explica en el Anexo 2)
for j in 1:Dim[2]
#Barremos toda la cadena
if Cest[j]==true
#Comprobamos si la columna es estable.
if Moda[j]==CadenaA[j]
global sumaE+=1
end
#Comparamos cada columna con la moda de esa columna en el archivo de learning.
else
global sumaI+=length(findall(seqs[:,j].==CadenaA[j]))
#Comparamos el aminoacido de esa columna con la columna entera del archivo de learning.
end
end
end
println(Limite,'\t',sumaE/(Nest*Ncad),'\t',sumaI/((Lcad-Nest)*Ncad*Dim[1]))
global sumaE = 0
global sumaI = 0
global j +=1
end
return Estables,Moda
end
```

5.5. Obtención de λ y Ω

5.5.1. Omega

Vamos a explicar cómo hallamos la covarianza pesada y la correlación pesada para poder hallar sus normas de Frobenius y a partir de ellas determinar Ω.

```
getnorm(cov) = norm(cov,2)
#Calcula la norma de Frobenius de una matriz.

function covvector(Seqs,weights)
#Calcula la matriz de covarianzas y la de correlaciones a partir de los pesos de cada secuencia y de las propias
secuencias en binario.
w=weights/sum(weights)
#Por comodidad normalizamos los pesos dividiendo entre Me.
we=StatsBase.weights(w)
#Es necesario cambiar el tipo de dato del vector de pesos para que cov() funcione
mean,cov=StatsBase.mean_and_cov(Seqs,we::AbstractWeights; corrected=false)
#Calcula la covarianza y media pesadas.
```

```
Desvi=DesviacionV(Seqs,mean,we)
#Calcula la desviacion estandar (Vector) teniendo en cuenta los
   pesos de cada cadena.
CorrelationM=Correlation(cov,Desvi)
#Devuelve la matriz de correlaciones.
return CorrelationM,cov
end

function Correlation(cov,Desvi)
#Calcula la matriz de correlaciones a partir de la matriz de
   covarianza y las desviaciones estandar calculadas en la funcion
   covvector. Esta hecho de esta manera porque tenemos filas enteras
    de 0 en las cuales la desviacion estandar es 0 y cualquier otra
   funcion de Julia que calcule esta matriz de forma vectorizada
   devuelve NaN (Not a Number) en lugar de
cero para el valor de ese elemento de la matriz de correlaciones.
Dimen=size(cov)
Corr=similar(cov)
for i in 1:Dimen[1]
if Desvi[i] == 0.0
Corr[:,i].=0.0
Corr[i,:].=0.0
else
Corr[:,i].=cov[:,i]/Desvi[i]
Corr[i,:].=cov[i,:]/Desvi[i]
end
end
return Corr
end

function DesviacionV(Seqs,Mean,WeightsV)
#Calcula la desviacion estandar pesada de cada una de las columnas.
Desv = Array{Float64}(undef,size(Seqs)[2])
for i in 1:size(Seqs)[2]
Desv[i]=StatsBase.std(Seqs[:,i],WeightsV;mean=Mean[i],corrected=
   false)
end
return Desv
end

const Omegarange = range(0.01,stop=0.99,length=100)
#Vamos a barrer desde 0.15 hasta 0.65 haciendo 100 pasos intermedios
   .

for Omega in Omegarange
w= weight(seqs,customOmega=Omega, memoize=true)
#Calculamos los pesos de cada secuencia.
Me=sum(w)
corr,cov=covvector(seqsB, w)
#Calculamos las matrices de correlacion y covarianza.
NormCov=getnorm(cov)
NormCorr=getnorm(corr)

#Calculamos las Normas de Frobenius.
```

```
end
```

5.5.2. Lambda

Veamos el método de minimizar d_1 y d_2.
Vamos a utilizar muchas de las funciones que hemos usado en el apartado de hallar Ω. Primero hemos de calcular la covarianza y la media pesadas del archivo de learning y después las de nuestro archivo de cadenas. Una vez calculadas, aplicamos la definición de d_1 y d_2.

```
function DifNorm(Sig,Omegat)
Mutest,Sigtest= Posterior(seqstest,
lambda = lambda,
dontcare = false,
customOmega = Omegat,
memoize = true)
#Funcion tomada de Ref4.

sigaux=sigtest-sigref
muaux=mutest-muref
#Calculamos el argumento para calcular d1 y d2.

return(norm(sigaux,2),norm(muaux,2))
#Devolvemos d1 y d2.
end
```

5.6. Hamiltoniano de frecuencias experimentales

En este anexo vamos a presentar el código utilizado para implementar el método de Monte Carlo siguiendo el hamiltoniano propuesto en la sección 3.2.4.

```
using LinearAlgebra
using DelimitedFiles
using Statistics
using Random
using NwiAnalizer

aminosdictD = ['A','C','D','E','F','G','H',
'I','K','L','M','N','P','Q',
'R','S','T','V','W','Y','-']

if length(ARGS) != 3
        println("./MejoraEnerAmin.jl Referencia Ncadenas Nsem")
        exit(0)
else
        global Nsem=parse(Int64, ARGS[3])
        global Ncadenas=parse(Int64, ARGS[2])
        global Intro = ARGS[1]
end
```

```
19
20 println("Numero de semillas = $Nsem",'\t',"Numero de cadenas =
       $Ncadenas")
21 println("Usando como learning $Intro")
22
23 global seqs = inputtoascii(Intro)
24 global Namin = length(aminosdictD)
25 global Nseqs = size(seqs)[1]
26 global Lcad = size(seqs)[2]
27
28 global Termaliza = 200 #PASOS DE TERMALIZACION A REALIZAR ANTES DE
       COMENZAR A MEDIR.
29 global Nbarrido = 50 #barrido ENTRE ESCRITURA Y ESCRITURA.
30 global Xcamb = 0.7 #PORC. Max de cadena a cambiar en cada barrido
31 global BetaSim = 1.0
32
33 function HallaFrec(seqs)
34 #Calcula la frecuencia de cada aminoacido en cada columna y un
       vector con cuantos
35 #distintos hay en cada columna.
36 MatFrec=Matrix{Float64}(undef,Namin,Lcad)
37 for i in 1:Lcad
38 for k in 1:Namin
39 MatFrec[k,i]=(length(findall(isequal(aminosdictD[k]),seqs[:, i]))/
       Nseqs)
40 end
41 end
42 return MatFrec
43 end
44
45 function CalculaMatE(MatFrec)
46 #Calcula la matriz con la energia de cada aminoacido en cada columna
       a partir de las frecuencias
47 #de aparicion de ambos.
48         MatEner=similar(MatFrec)
49         for i in 1:Lcad
50                 for k in 1:Namin
51                         if MatFrec[k,i]==0.0
52                                 MatEner[k,i]=100.0 #Maximo de
                                       energia arbitrario para los
                                       aminoacidos con frec. 0
53                         else
54                                 MatEner[k,i]=log(1/MatFrec[k,i])
55                         end
56                 end
57         end
58         return MatEner
59 end
60
61 function CalculaEnerCad(Cadena,MatEner)
62 #Calcula la energia de una cadena
63         global Energia=0.0
64         for i in 1:Lcad
65                 Energia+=MatEner[Cadena[i],i]
```

```
66          end
67          return Energia
68  end
69
70  function Escribe(Cadcand)
71          Srt=string.(Cadcand[1])
72          for k in 2:length(Cadcand)
73                  Srt=string(Srt,Cadcand[k])
74          end
75          writedlm(fichero,[Srt])
76  end
77
78  function AceptaCamb(Eaux,Ecand,Cadcand,Cadaux)
79  #Calculamos la exponencial de la diferencia de energias para ver si
        aceptamos el cambio.
80          C=exp(-BetaSim*(Eaux-Ecand))
81          if(C>rand())
82                  Cadcand=copy(Cadaux)
83                  Ecand=copy(Eaux)
84          else
85                  Cadaux=copy(Cadcand)
86          end
87          return Cadcand,Ecand
88  end
89
90  function NumaLetras(Cadcand)
91  #Transforma la cadena de numeros a letras para poder almacenarla
        para que los biologos entiendan algo
92          CadCandLetras=Array{Char}(undef,Lcad)
93          for j in 1:Lcad
94                  CadCandLetras[j]=aminosdictD[Cadcand[j]]
95          end
96                  return CadCandLetras
97          end
98
99  function OrdenCambio()
100         Porccamb = Integer(trunc(Lcad*Xcamb))
101         PosCamb = randperm!(Vector{Int}(undef, Porccamb))
102         AminCamb = rand(1:Namin,Porccamb)
103         return PosCamb,AminCamb
104 end
105
106 function barrido(Cadcand,Ecand,PosCamb,AminCamb,i)
107 #Realiza un barrido
108         Cadaux=copy(Cadcand)
109         for j in 1:length(PosCamb)
110                 Cadaux[PosCamb[j]]=AminCamb[j]
111                 Eaux=CalculaEnerCad(Cadaux,MatEner)
112                 Cadcand,Ecand=AceptaCamb(Eaux,Ecand,Cadcand,Cadaux)
113                 Cadaux=copy(Cadcand)
114         end
115         return Cadcand
116 end
117
```

```
118 function Pasos(PasosTot)
119         Cadcand=rand(1:Namin,Lcad)
120         CandP=copy(Cadcand)
121         for i in 1:PasosTot
122                 Ecand = CalculaEnerCad(Cadcand,MatEner)
123                 PosCamb,AminCamb=OrdenCambio()
124                 CadN=barrido(Cadcand,Ecand,PosCamb,AminCamb,i)
125                 if (i>Termaliza && i%Nbarrido==0)
126                         CadL=NumaLetras(CadN)
127                         Escribe(CadL)
128                 end
129                 Cadcand=copy(CadN)
130         end
131 end
132
133
134 println("Librerias y funciones cargadas")
135
136 global fichero=open("$(Integer(Ncadenas))C$(Integer(Nsem))S$(Integer
        (Nbarrido))W.txt","w")
137
138 println(Calculamos matrices de energia y de frecuencias)
139 MatFrec=HallaFrec(seqs)
140 MatEner=CalculaMatE(MatFrec)
141
142 PasosTot =Termaliza + Integer(trunc(Ncadenas*Nbarrido/Nsem))
143 println("Pasos totales: $(PasosTot*Nsem)")
144
145 println("Comenzamos a termalizar")
146
147 for j in 1:Nsem
148         println("Utilizando semilla $j de $Nsem")
149         Pasos(PasosTot)
150 end
```